

第一次上机报告

2.1 熟悉Linux环境下pthread编程，完成示例代码的编译与执行

nosync-ex.c

使用 `vim nosync-ex.c` 创建 C 语言脚本，并输入以下代码：

```
#include <pthread.h>
#include <stdio.h>

int sum = 0;

void* thread(void*) {
    int i;
    for (i = 0; i < 1000000; i++)
        sum++;

    return NULL;
}

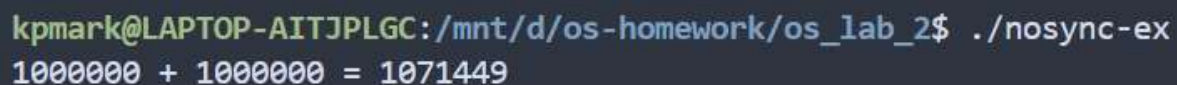
int main(void) {
    pthread_t tid1, tid2;

    pthread_create(&tid1, NULL, thread, NULL);
    pthread_create(&tid2, NULL, thread, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("1000000 + 1000000 = %d\n", sum);
    return 0;
}
```

然后，使用 `gcc -o nosync-ex nosync-ex.c` 命令编译 C 语言脚本，并再同一目录下使用 `./nosync-ex` 命令执行可执行文件，得到如下结果：



```
kpmark@LAPTOP-AITJPLGC:/mnt/d/os-homework/os_lab_2$ ./nosync-ex
1000000 + 1000000 = 1071449
```

答案错误，是因为 `sum` 变量是全局变量，在各个线程中被争用，导致 `sum` 求和循环的判断条件提前到达，提前退出循环，所以最终 `sum` 的值小于 2000000。

mutex-ex

使用 `vim mutex-ex.c` 创建 C 语言脚本，并输入以下代码：

```
#include <pthread.h>
#include <stdio.h>

int sum = 0;
```

```

pthread_mutex_t mutex;

void* thread(void*) {
    int i;
    for (i = 0; i < 1000000; i++) {
        pthread_mutex_lock(&mutex);
        sum++;
        pthread_mutex_unlock(&mutex);
    }
}

// 使用互斥锁

int main(void) {
    pthread_t tid1, tid2;

    pthread_mutex_init(&mutex, NULL);

    pthread_create(&tid1, NULL, thread, NULL);
    pthread_create(&tid2, NULL, thread, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("1000000 + 1000000 = %d\n", sum);

    pthread_mutex_destroy(&mutex);
    return 0;
}

```

然后，使用 `gcc -o mutex-ex mutex-ex.c` 命令编译 C 语言脚本，并再同一目录下使用 `./mutex-ex` 命令执行可执行文件，得到如下结果：

```

kpmark@LAPTOP-AITJPLGC:/mnt/d/os-homework/os_lab_2$ ./mutex-ex
1000000 + 1000000 = 2000000

```

这个脚本使用互斥锁实现了对变量 `sum` 的访问控制，一个线程在使用 `sum` 时另外一个线程必须等待，使得两个线程同步。

sem-ex

使用 `vim sem-ex.c` 创建 C 语言脚本，并输入以下代码：

```

#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

int sum = 0;
sem_t sem;

void* thread(void*) {
    int i;
    for (i = 0; i < 1000000; i++) {
        sem_wait(&sem)
        sum++;
        sem_post(&sem);
    }
}

```

```

int main(void) {
    pthread_t tid1, tid2;

    sem_init(&sem, 0, 1);

    pthread_create(&tid1, NULL, thread, NULL);
    pthread_create(&tid2, NULL, thread, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("1000000 + 1000000 = %d\n", sum);

    sem_destroy(&sem);
    return 0;
}

```

然后，使用 `gcc -o sem-ex sem-ex.c` 命令编译 C 语言脚本，并再同一目录下使用 `./sem-ex` 命令执行可执行文件，得到如下结果：

```

kpmark@LAPTOP-AITJPLGC:/mnt/d/os-homework/os_lab_2$ ./sem-ex
1000000 + 1000000 = 2000000

```

这个脚本使用信号量控制变量 `sum` 的使用，通过 `sem_wait(&sem)` 占用信号量，控制另一个线程的访问，使用结束后再使用 `sem_post(&sem)` 接触占用。

2.2 基于示例中涉及到的线程同步API，实现生产者消费者问题

在 `test.c` 中编写如下代码：

```

#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <unistd.h>

int sum = 0;

sem_t empty;
sem_t full;
pthread_mutex_t mutex;

void* producer(void* arg) {
    for (int i = 1; i <= 10; i++) {
        sem_wait(&empty);

        pthread_mutex_lock(&mutex);

        sum++;
        printf("生产者：生产数据，sum = %d\n", sum);

        pthread_mutex_unlock(&mutex);

        sem_post(&full);
    }
}

```

```

        return NULL;
    }

    void* consumer(void* arg) {
        for (int i = 1; i <= 10; i++) {
            sem_wait(&full);

            pthread_mutex_lock(&mutex);

            sum--;
            printf("消费者：消费数据，sum = %d\n", sum);

            pthread_mutex_unlock(&mutex);

            sem_post(&empty);
        }
        return NULL;
    }

    int main() {
        pthread_t prod, cons;

        pthread_mutex_init(&mutex, NULL);
        sem_init(&empty, 0, 1);
        sem_init(&full, 0, 0);

        printf("开始...\n");

        pthread_create(&prod, NULL, producer, NULL);
        pthread_create(&cons, NULL, consumer, NULL);

        pthread_join(prod, NULL);
        pthread_join(cons, NULL);

        pthread_mutex_destroy(&mutex);
        sem_destroy(&empty);
        sem_destroy(&full);

        printf("结束\n");
        return 0;
    }
}

```

1. 生产者流程：

- 等待空位(`sem_wait(&empty)`)
- 获取互斥锁(`pthread_mutex_lock(&mutex)`)
- 增加数据(`sum++`)
- 释放互斥锁(`pthread_mutex_unlock(&mutex)`)
- 标记有数据可用(`sem_post(&full)`)

2. 消费者流程：

- 等待数据(`sem_wait(&full)`)
- 获取互斥锁(`pthread_mutex_lock(&mutex)`)
- 消费数据(`sum--`)
- 释放互斥锁(`pthread_mutex_unlock(&mutex)`)
- 标记有空位可用(`sem_post(&empty)`)

运行结果如下：

```
kpmark@LAPTOP-AITJPLGC:/mnt/d/os-homework/os_lab_2$ ./test
开始...
生产者：生产数据, sum = 1
消费者：消费数据, sum = 0
生产者：生产数据, sum = 1
消费者：消费数据, sum = 0
生产者：生产数据, sum = 1
消费者：消费数据, sum = 0
生产者：生产数据, sum = 1
消费者：消费数据, sum = 0
生产者：生产数据, sum = 1
消费者：消费数据, sum = 0
生产者：生产数据, sum = 1
消费者：消费数据, sum = 0
生产者：生产数据, sum = 1
消费者：消费数据, sum = 0
生产者：生产数据, sum = 1
消费者：消费数据, sum = 0
生产者：生产数据, sum = 1
消费者：消费数据, sum = 0
生产者：生产数据, sum = 1
消费者：消费数据, sum = 0
生产者：生产数据, sum = 1
消费者：消费数据, sum = 0
结束
```

代码阅读理解

3.1 pthread-ex01

这段代码创建了一个线程，该线程立即退出并返回值 42，主线程通过 `pthread_join` 等待子线程结束并获取其返回值，然后将该返回值打印出来。

3.2 pthread-ex02

这段代码创建了一个线程，该线程通过调用 `exit(42)` 直接终止整个进程（而不仅是线程），导致主程序立即结束，而不会执行到 `pthread_join` 及后续的打印语句。

3.2 pthread-ex03

这段代码创建了两个线程：thread 线程尝试返回变量 `i` 的值 (42)，thread2 线程将变量 `i` 修改为 0 并返回 31，主线程先等待 thread 结束并获取其返回值替换 `i`，然后等待 thread2 结束，最后打印 `i` 的值（实际输出取决于线程执行顺序，可能是 42 或 31）。

3.2 pthread-ex04

这段代码创建了一个线程，该线程将自身设置为分离状态 (detached) 后退出并返回值 42，由于分离状态的线程资源会自动回收且无法被 `join`，所以主线程的 `pthread_join` 调用会失败，`i` 的值不会被更新为 42，最终打印的可能仍是初始值 0。

3.2 pthread-ex05

这段代码创建了两个线程：thread2 线程将全局变量i修改为 31，thread 线程打印全局变量 i 的值，由于线程创建顺序和执行时机的不确定性，输出可能是 42（如果 thread 先于 thread2 执行）或 31（如果 thread2 先于 thread 执行）。

3.2 pthread-ex06

这段代码循环创建两个线程，每个线程被传递一个指向动态分配内存的指针，线程函数从指针获取线程 ID 值并打印，然后释放该内存，主线程等待两个线程结束，输出应为 Thread 0 和 Thread 1（顺序不确定）。

3.2 pthread-ex07

这段代码循环创建两个线程，每个线程共享同一个变量 i 的地址，由于线程执行和循环迭代的速度不确定，当线程实际访问变量 i 时，其值可能已经改变，导致可能输出 Thread 1 两次，或者 Thread 0 和 Thread 1，甚至 Thread 2。

解决理发师问题

代码如下：

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

sem_t barber_ready;
sem_t customer_ready;
sem_t mutex;

int waiting_customers = 0;
int chairs = 3;

void* barber_function(void* arg) {
    while (1) {
        printf("理发师：睡觉中...\n");
        sem_wait(&customer_ready);

        sem_wait(&mutex);
        waiting_customers--;
        sem_post(&mutex);

        printf("理发师：为顾客理发中...\n");
        sleep(2);

        printf("理发师：理发完成\n");
        sem_post(&barber_ready);
    }
    return NULL;
}

void* customer_function(void* arg) {
    int id = *((int*)arg);
    free(arg);
```

```

sleep(rand() % 5);

printf("顾客 %d: 到达理发店\n", id);

sem_wait(&mutex);
if (waiting_customers < chairs) {
    waiting_customers++;
    printf("顾客 %d: 等待理发, 当前等待人数: %d\n", id, waiting_customers);
    sem_post(&mutex);

    sem_post(&customer_ready);

    sem_wait(&barber_ready);
    printf("顾客 %d: 理发完成, 离开理发店\n", id);
} else {
    printf("顾客 %d: 没有空位, 离开理发店\n", id);
    sem_post(&mutex);
}

return NULL;
}

int main() {
    pthread_t barber_thread;
    pthread_t customer_threads[10];

    sem_init(&barber_ready, 0, 0);
    sem_init(&customer_ready, 0, 0);
    sem_init(&mutex, 0, 1);

    pthread_create(&barber_thread, NULL, barber_function, NULL);
    for (int i = 0; i < 10; i++) {
        int* id = malloc(sizeof(int));
        *id = i + 1;
        pthread_create(&customer_threads[i], NULL, customer_function, id);
    }

    for (int i = 0; i < 10; i++) {
        pthread_join(customer_threads[i], NULL);
    }

    printf("所有顾客已处理完毕, 程序结束\n");
    return 0;
}

```

运行结果如下图:

```
kpmark@LAPTOP-AITJPLGC:/mnt/d/os-homework/os_lab_2$ gcc -o test2 test2.c
kpmark@LAPTOP-AITJPLGC:/mnt/d/os-homework/os_lab_2$ ./test2
理发师: 睡觉中...
顾客 5: 到达理发店
顾客 5: 等待理发, 当前等待人数: 1
理发师: 为顾客理发中...
顾客 6: 到达理发店
顾客 6: 等待理发, 当前等待人数: 1
顾客 2: 到达理发店
顾客 2: 等待理发, 当前等待人数: 2
顾客 7: 到达理发店
顾客 7: 等待理发, 当前等待人数: 3
顾客 10: 到达理发店
顾客 10: 没有空位, 离开理发店
顾客 3: 到达理发店
顾客 3: 没有空位, 离开理发店
理发师: 理发完成
理发师: 睡觉中...
理发师: 为顾客理发中...
顾客 5: 理发完成, 离开理发店
顾客 8: 到达理发店
顾客 8: 等待理发, 当前等待人数: 3
顾客 1: 到达理发店
顾客 1: 没有空位, 离开理发店
顾客 4: 到达理发店
顾客 4: 没有空位, 离开理发店
顾客 9: 到达理发店
顾客 9: 没有空位, 离开理发店
理发师: 理发完成
理发师: 睡觉中...
理发师: 为顾客理发中...
顾客 6: 理发完成, 离开理发店
理发师: 理发完成
理发师: 睡觉中...
理发师: 为顾客理发中...
顾客 2: 理发完成, 离开理发店
理发师: 理发完成
理发师: 睡觉中...
理发师: 为顾客理发中...
顾客 7: 理发完成, 离开理发店
理发师: 理发完成
理发师: 睡觉中...
顾客 8: 理发完成, 离开理发店
所有顾客已处理完毕, 程序结束
```

Git 记录

运行命令 `git log`, 结果如下:


```
PS D:\os-homework> git log
commit 9fbc2e17aa0ce835e247495d0fea187c5d45e5bf (HEAD -> main)
Author: kpmark <2585050765@qq.com>
Date: Tue May 13 22:21:29 2025 +0800
```

新增理发师问题test2, 完成实验报告

```
commit 5e5c1ee9f1b14588947910304040adfd74ccd71
Author: kpmark <2585050765@qq.com>
Date: Tue May 13 18:53:39 2025 +0800
```

生产者作业test

```
commit 61d394948c5366f303dc98c55d44d871333b1fbc
Author: kpmark <2585050765@qq.com>
Date: Tue May 13 18:25:37 2025 +0800
```

新增部分注释

```
commit bec7e277b505f26e7913b576ae1fd2e3264dc08d
Author: kpmark <2585050765@qq.com>
Date: Tue May 13 18:19:48 2025 +0800
```

新增sem-ex

```
commit 1916d3a00604af478037a90a71c5832356a2b386
Author: kpmark <2585050765@qq.com>
Date: Tue May 13 18:16:54 2025 +0800
```

新增mutex-ex

```
commit 426e3920569cec61d3de338e0e47074434eaaf3a
Author: kpmark <2585050765@qq.com>
Date: Tue May 13 18:11:46 2025 +0800
```

新增nosync-ex

可视化界面如下:



作业仓库地址为[markzhang12345/os-homework](https://github.com/markzhang12345/os-homework): [大连理工大学操作系统课程作业](#)